CALL

CALL [INTERFACE4] *operand1* **[USING]** **[***operand2***]...** *128*

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
Operand1	C	S				A												yes	no
Operand2	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G		yes	yes

- CALL on Mainframe Computers
- Part I: CALL under OpenVMS, UNIX and Windows
- INTERFACE4
- Part II: CALL under OpenVMS, UNIX and Windows

CALL on Mainframe Computers

- Function
- Program Name - operand1
- Parameters - operand2
- Return Code
- Register Usage
- Boundary Alignment
- Adabas Calls
- Direct/Dynamic Loading
- Example
- Linkage Conventions
- Calling a PL/I Program

Function

The CALL statement is used to call an external program written in another standard programming language from a Natural program and then return to the next statement after the CALL statement.

The called program may be written in any programming language which supports a standard CALL interface. Multiple CALL statements to one or more external programs may be specified.

A CALL statement may be issued within a program to be executed under control of a TP monitor, provided that the TP monitor supports a CALL interface.

Program Name - *operand1*

The name of the program to be called *(operand1)* can be specified as a constant or - if different programs are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A program name must be placed left-justified in the variable.

Parameters - *operand2*

The CALL statement may contain up to 128 parameters *(operand2)*, unless the INTERFACE4 option is used. In that case, there is no limit as to how many parameters may be used. Standard linkage register conventions are used. One address is passed in the parameter list for each parameter field specified.

If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified.

Note:
The internal representation of positive signs of packed numbers is changed to the value specified by the PSIGNF parameter of the NTCMPO macro **before** control is passed to the external program.

Return Code

The condition code of any called program (content of register 15 upon return to Natural) may be obtained by using the Natural system function RET.

Example:

```
...
RESET #RETURN(B4)
CALL 'PROG1'
IF RET ('PROG1') > #RETURN
  WRITE 'ERROR OCCURRED IN PROGRAM1'
END-IF
...
```

Register Usage

Register	Contents
R1	Address pointer to the parameter address list.
R2	Address pointer to the field (parameter) description list. The field description list contains information on the first 128 fields passed in the parameter list. Each description is a 4-byte entry containing the following information: - the 1st byte contains the type of variable (A,B,...) If field type is "N" or "P": - the 2nd byte contains the total number of digits; - the 3rd byte contains the number of digits before the decimal point; - the 4th byte contains the number of digits after the decimal point. all other field types: - the 2nd byte is unused; - the 3rd-4th byte contain the length of field.
R3	Address pointer to list of field lengths. This list contains the length of each field passed in the parameter list. In the case of an array, the length is the sum of the individual occurrences' lengths.
R13	Address of 18-word save area.
R14	Return address.
R15	Entry address/return code.

Boundary Alignment

The Natural data area, in which all user-defined variables are stored, always begins on a double-word boundary.

If DEFINE DATA is used, all data blocks (for example, LOCAL, GLOBAL blocks) are double-word aligned, and all structures (level 1) are full-word aligned.

Alignment within the data area is the responsibility of the user and is governed by the order in which variables are defined to Natural.

Adabas Calls

A called program may contain a call to Adabas. The called program must not issue an Adabas open or close command. Adabas will open all database files referenced. If Adabas exclusive (EXU) update mode is to be used, the Natural profile parameter OPRB must be used in order to open all referenced files. Before you attempt to use EXU update mode, you should consult your Natural administrator.

Direct/Dynamic Loading

The called program may either be directly linked to the Natural nucleus (that is, the program is specified with the CSTATIC parameter in the Natural parameter module; see also the Natural Operations documentation for Mainframes, or it may be loaded dynamically the first time it is called. If it is to be loaded dynamically, the load module library containing the called program must be concatenated to the Natural load library in the Natural execution JCL or in the appropriate TP-monitor program library. Ask your Natural administrator for additional information.

Example

The example on the next page shows a Natural program which calls the COBOL program "TABSUB" for the purpose of converting a country code into the corresponding country name. Two parameter fields are passed by the Natural program to TABSUB: the first parameter is the country code, as read from the database; the second parameter is used to return the corresponding country name.

Calling Natural Program:

```
* EXAMPLE 'CALEX1': CALL PROGRAM 'TABSUB'
* ****************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 COUNTRY
1 #COUNTRY (A3)
1 #COUNTRY-NAME (A15)
END-DEFINE
*
MOVE EDITED '19550701' to #FIND-FROM (EM=YYYYMMDD)
MOVE EDITED #19550731' to #FIND-TO   (EM=YYYYMMDD)
*
FIND EMPLOY-VIEW WITH BIRTH = #FIND-FROM THRU #FIND-TO
  MOVE COUNTRY TO #COUNTRY
  CALL 'TABSUB' #COUNTRY #COUNTRY-NAME
  DISPLAY NAME BIRTH (EM=YYYY-MM-DD) #COUNTRY-NAME
END-FIND
END
```


Called COBOL program "TABSUB":

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PROVIDES THE COUNTRY NAME
        FOR A GIVEN COUNTRY CODE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
01 COUNTRY-CODE  PIC X(3).
01 COUNTRY-NAME  PIC X(15).
PROCEDURE DIVISION USING COUNTRY-CODE COUNTRY-NAME.
P-CONVERT.
   MOVE SPACES TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'BLG' MOVE 'BELGIUM' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'DEN' MOVE 'DENMARK' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'FRA' MOVE 'FRANCE' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'GER' MOVE 'GERMANY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'HOL' MOVE 'HOLLAND' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'ITA' MOVE 'ITALY' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'SPA' MOVE 'SPAIN' TO COUNTRY-NAME.
   IF COUNTRY-CODE = 'UK'  MOVE 'UNITED KINGDOM' TO COUNTRY-NAME.
P-RETURN.
GOBACK.
```

Linkage Conventions

- CALL using Com-plete
- CALL using CICS
- Return Codes under CICS
- Example using CICS

Standard linkage register notation is used in batch mode. Each TP monitor has its own conventions. These conventions must be followed; otherwise, unpredictable results could occur. The following sections describe conventions that apply for the supported TP monitors.

CALL using Com-plete

The called program must reside in the Com-plete online load library. This allows Com-plete to load the program dynamically. The Com-plete utility ULIB may be used to catalog the program.

CALL using CICS

The called program must reside in either a load module library concatenated to the CICS library or the DFHRPL library. The program must also have a PPT entry in the operating PPT so that CICS can locate the program and load it.

The linkage convention passes the parameter list address followed by the field description list address in the first fullwords of the TWA and the COMMAREA.The parameter FLDLEN in the NCIPARM parameter module controls if the field length list is also passed (by default, it is not passed). The COMMAREA length (8 or 12) reflects the number of list addresses passes (2 or 3). The last list address is indicated by the high-order bit being set. The user must ensure addressability to the TWA or to the COMMAREA respectively. This is only required if the user program has not been defined to Natural as a static or directly linked program, in which case the pointer to the parameter list is passed via register 1, the pointer to the description list via register 2, and the pointer to the field length list via register 3.

If you wish the parameter values themselves, rather than the address of their address list, to be passed in the COMMAREA, issue the Natural terminal command %P=C before the call.

Normally, when a Natural programs calls a non-Natural program and the called program issues a conversational terminal I/O, the Natural thread is blocked until the user has entered data. To prevent the Natural thread from being blocked, the terminal command %P=V can be used

Normally, when a Natural program calls a non-Natural program under CICS, the call is accomplished by an "EXEC CICS LINK" request. If standard linkage is to be used for the call instead, issue the terminal command %P=S (In this case, the called program must adhere to standard linkage conventions with standard register usage).

In 31-bit-mode environments the following applies: if a program linked with AMODE=24 is called and the threads are above 16 MB, a "call by value" will be done automatically, that is, the specified parameters which are to be passed to the called program will be copied below 16 MB.

Return Codes under CICS

CICS itself does not support condition codes for a call with CICS conventions (EXEC CICS LINK). However, the Natural CICS Interface supports return codes for the CALL statement: When control is returned from the called program, Natural checks whether the first fullword of the COMMAREA has changed. If it has, its new content will be taken as the return code. If it has not changed, the first fullword of the TWA will be checked and its new content taken as the return code. If neither of the two fullwords has changed, the return code will be "0".

Note:
When parameter values are passed in the COMMAREA (%P=C), the return code is always "0".

Example using CICS:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSUB.
REMARKS. THIS PROGRAM PERFORMS A TABLE LOOK-UP AND
         RETURNS A TEXT MESSAGE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MSG-TABLE.
   03  FILLER       PIC X(15) VALUE 'MESSAGE1       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE2       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE3       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE4       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE5       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE6       '.
   03  FILLER       PIC X(15) VALUE 'MESSAGE7       '.
01 TAB REDEFINES MSG-TABLE.
   03  MESSAGE OCCURS 7 TIMES PIC X(15).
LINKAGE SECTION.
01 TWA-DATA.
   03 PARM-POINTER USAGE IS POINTER.
01 PARM-LIST.
   03 DATA-LOC-IN  USAGE IS POINTER.
   03 DATA-LOC-OUT USAGE IS POINTER.
01 INPUT-DATA.
   03  INPUT-NUMBER      PIC 99.
01 OUTPUT-DATA.
   03  OUTPUT-MESSAGE    PIC X(15).
PROCEDURE DIVISION.
100-INIT.
    EXEC CICS ADDRESS TWA(ADDRESS OF TWA-DATA) END-EXEC.
    SET ADDRESS OF PARM-LIST   TO PARM-POINTER.
    SET ADDRESS OF INPUT-DATA  TO DATA-LOCIN.
    SET ADDRESS OF OUTPUT-DATA TO DATA-LOC-OUT.
200-PROCESS.
    MOVE MESSAGE (INPUT-NUMBER) TO OUTPUT-MESSAGE.
300-RETURN.
    EXEC CICS RETURN END-EXEC.
400-DUMMY.
    GO-BACK.
```

Calling a PL/I Program

- Example of Calling a PL/I Program:
- Example of Calling a PL/I Program which is Operating under CICS

A called program written in PL/I requires the following additional procedures:

- The ENTRY PLICALLA statement must be provided when the program is link-edited. This statement causes the PL/I load module to receive control as a sub-program (that is, a called program).
  If the PL/I program is to be called recursively, you may also use the program NATPLICA, which is contained in the Natural source library. NATPLICA is an example of how a PL/I program can be called recursively from a Natural program without causing any storage bottlenecks (for further details, please refer to the comments in the program NATPLICA itself). A complete description of the ENTRY PLICALLA statement and further information on how to call a PL/I program can be found in the relevant IBM PL/I documentation.
- Since the parameter list is a standard list and is not an argument list being passed from another PL/I program, the addresses passed do not point at a LOCATOR DESCRIPTOR. This problem may be resolved by defining the parameter fields as arithmetic variables. This causes PL/I to treat the parameter list as addresses of data instead of addresses of LOCATOR DESCRIPTOR control blocks.

The technique suggested for defining the parameter fields is illustrated in the following example:

```
PLIPROG: PROC(INPUT_PARM_1, INPUT_PARM_2) OPTIONS(MAIN);
      DECLARE (INPUT_PARM_1, INPUT_PARM_2) FIXED;
      PTR_PARM_1 = ADDR(INPUT_PARM_1);
      PTR_PARM_2 = ADDR(INPUT_PARM_2);
      DECLARE FIRST_PARM        PIC '99'   BASED (PTR_PARM_1);
      DECLARE SECOND_PARM       CHAR(12)   BASED (PTR_PARM_2);
```

Each parameter in the input list should be treated as a unique element. The number of input parameters should exactly match the number being passed from the Natural program. The input parameters and their attributes must match the Natural definitions or unpredictable results may occur. For additional information on passing parameters in PL/I, see the relevant IBM PL/I documentation.

Example of calling a PL/I Program:

```
/* EXAMPLE 'CALEX2': CALL PROGRAM 'NATPLI'
/**********************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 AREA-CODE
  2 REDEFINE AREA-CODE
    3 #AC(N1)
1 #INPUT-NUMBER (N2)
1 #OUTPUT-COMMENT (A15)
END-DEFINE
/**********************************************************************
READ EMPLOY-VIEW IN LOGICAL SEQUENCE BY NAME
                 STARTING FROM 'WAGNER'
 MOVE ' ' TO #OUTPUT-COMMENT
 MOVE #AC TO #INPUT-NUMBER
CALL 'NATPLI' #INPUT-NUMBER #OUTPUT-COMMENT
END-READ
END
```

```
NATPLI:  PROC(PARM_COUNT, PARM_COMMENT) OPTIONS(MAIN);
    /*                                            */
    /* THIS PROGRAM ACCEPTS AN INPUT NUMBER       */
    /* AND TRANSLATES IT TO AN OUTPUT CHARACTER   */
    /* STRING FOR PLACEMENT ON THE FINAL          */
    /* NATURAL REPORT                             */
    /*                                            */
    /*                                            */
   DECLARE PARM_COUNT, PARM_COMMENT  FIXED;
   DECLARE ADDR BUILTIN;
   COUNT_PTR = ADDR(PARM_COUNT);
   COMMENT_PTR = ADDR(PARM_COMMENT);
   DECLARE  INPUT_NUMBER    PIC '99' BASED (COUNT_PTR);
   DECLARE  OUTPUT_COMMENT  CHAR(15) BASED (COMMENT_PTR);
   DECLARE COMMENT_TABLE(9) CHAR(15) STATIC INITIAL
      ('COMMENT1   ',
       'COMMENT2   ',
       'COMMENT3   ',
       'COMMENT4   ',
       'COMMENT5   ',
       'COMMENT6   ',
       'COMMENT7   ',
       'COMMENT8   ',
       'COMMENT9   ');
    OUTPUT_COMMENT = COMMENT_TABLE(INPUT_NUMBER);
    RETURN;
END NATPLI;
```

Example of Calling a PL/I Program which is Operating under CICS:

```
/* EXAMPLE 'CALEX3': CALL PROGRAM 'CICSP'
/******************************************
DEFINE DATA LOCAL
1 #MESSAGE (A10) INIT <' '>
END-DEFINE
/******************************************
CALL 'CICSP' #MESSAGE
DISPLAY #MESSAGE
/******************************************
END
```

```
CICSP: PROCEDURE OPTIONS (MAIN REENTRANT);
       DCL  1        TWA_ADDRESS    BASED(TWA_POINTER);
            2        LIST_ADDRESS   POINTER;
       DCL  1 PTR_TO_LIST           BASED(LIST_ADDRESS);
            2 PARM_01               POINTER;
       DCL MESSAGE CHAR(10) BASED(PARM_01);
       EXEC CICS ADDRESS TWA(TWA_POINTER);
       MESSAGE='SUCCESS'; EXEC CICS RETURN; END CICSP;
```


Part I: CALL under OpenVMS, UNIX and Windows

- Function
- Name of Called Function (operand1)
- Parameters (operand2)

Function

The CALL statement is used to call an external function written in another standard programming language from a Natural program and then return to the next statement after the CALL statement.

The called function may be written in any programming language which supports a standard CALL interface. Multiple CALL statements to one or more external functions may be specified.

Name of Called Function - *operand1*

The name of the function to be called *(operand1)* may be specified as a constant or - if different functions are to be called dependent on program logic - as an alphanumeric variable of length 1 to 8. A function name must be placed left-justified in the variable.

Parameters - *operand2*

The CALL statement may contain up to 128 parameters *(operand2)*. One address is passed to the external function in the parameter list for each parameter field specified.

If a group name is used, the group is converted to individual fields; that is, if a user wishes to specify the beginning address of a group, the first field of the group must be specified.

Note:
If an application-independent variable (AIV) or context variable is passed as a parameter to a user exit, the following restriction applies: if the user exit invokes a Natural subprogram which creates a new AIV or context variable, the parameter is invalid after the return from the subprogram. This is true regardless of whether the new AIV/context variable is created by the subprogram itself or by another object invoked directly or indirectly by the subprogram.

INTERFACE4

- INTERFACE 4 - External 3GL Program Interface
- Operand Structure for Interface4
- INTERFACE4 Parameter Access
- Exported Functions

Note:
The INTERFACE4 option is not available on mainframe computers.

The keyword INTERFACE4 specifies the type of the interface that is used for the call of the external program. This keyword is optional. If this keyword is specified, the interface, which is defined as ’Interface4’, is used for the call of the external program. The following table lists the differences between the CALL statement used with INTERFACE4 and the one used without INTERFACE4:

	CALL statement without keyword INTERFACE4	Call statement with keyword INTERFACE4
number of parameters possible	128	unlimited (32767)
maximum data size of one parameter	64 K	1 GB
retrieve array information	no	yes
support of large and dynamic operands	no	yes
parameter access via API	no	yes

INTERFACE4 - External 3GL Program Interface

The interface of the external 3GL program is defined as follows, when the INTERFACE4 is specified with the Natural CALL statement:

```
NATFCT functionname (numparm, parmhandle, traditional)
```

USR_WORD	numparm;	16 bit unsigned short value, containing the total number of transferred operands (operand2)
void	*parmhandle;	Pointer to the parameter passing structure.
void	*traditional;	Check for interface type (if it’s not a NULL pointer it’s the traditional CALL interface)

Operand Structure for Interface4

The operand structure of Interface4 is named ’parameter_description’ and is defined as follows. The structure is delivered with the header file natuser.h.

struct parameter_description		
void *	address	address of the parameter data, not aligned, realloc() and free() are not allowed
int	format	field data type: NCXR_TYPE_ALPHA, etc. (natuser.h)
int	length	length (before decimal point, if applicable)
int	precision	length after decimal point (if applicable)
int	byte_length	length of field in bytes int dimension number of dimensions (0 to IF4_MAX_DIM)
int	dimensions	number of dimensions (0 to IF4_MAX_DIM)
int	length_all	total data length of array in bytes
int	flags	several flag bits combined by bitwise OR, meaning: IF4_FLG_PROTECTED the parameter is write protected, IF4_FLG_DYNAMIC the parameter is a dynamic variable, IF4_FLG_NOT_CONTIGUOUS the array elements are not contiguous (have spaces between them), IF4_FLG_AIV is an application-independent variable
int	occurrences[IF4_MAX_DIM]	array occurrences in each dimension
int	indexfactors[IF4_MAX_DIM]	array indexfactors for each dimension
void *	dynp	reserved for internal use

The address of an array element (i,j,k) is computed as follows (especially if the array elements are not contiguous):

elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2].
If the array has less than 3 dimensions, leave out the last terms.

INTERFACE4 - Parameter Access

A set of functions is available to be used for the access of the parameters. The process flow is as follows. The 3GL program is called via the CALL statement with the INTERFACE4 option, and the parameters are passed to the 3GL program as described above. The 3GL program can now use the exported functions of Natural, to retrieve either the parameter data itself, or information about the parameter, like format, length, array information, etc. The exported functions can also be used to pass back parameter data. With this technique a parameter access is guaranteed to avoid memory overwrites done by the 3GL program. (Natural's data is safe: memory overwrites within the 3GL program's data are still possible).

Exported Functions

- Get parameter information
- Get parameter data
- Write back operand data

Get parameter information

This function is used by the 3GL program to receive all necessary information from any parameter. This information is returned in the struct parameter_description, which is documented above.

Prototype:

```
int ncxr_get_parm_info ( int parmnum, void *parmhandle, struct parameter_description *descr );
```

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	the pointer to the internal parameter structure
descr	address of a struct parameter_description
return	0: OK -1 illegal parameter number -2 internal error -7 interface version conflict

Get parameter data:

This function is used by the 3GL program to get the data from any parameter. Natural identifies the parameter by the given parameter number and writes the parameter data to the given buffer address with the given buffer size. If the parameter data is longer than the given buffer size, Natural will truncate the data to the given length. The external 3GL program can make use of the function ncxr_get_parm_info, to request the length of the parameter data. There are two functions to get parameter data: ncxr_get_parm gets the whole parameter (even if the parameter is an array), whereas ncxr_get_parm_array gets the specified array element.

If no memory of the indicated size is allocated for "buffer" by the 3GL program (dynamically or statically) results of the operation are unpredictable. Natural will only check for a null pointer.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_get_parm( int parmnum, void *parmhandle, int buffer_length, void *buffer )
```

```
int ncxr_get_parm_array( int parmnum, void *parmhandle, int buffer_length, void *buffer, int *indexes )
```

This function is identical to ncxr_get_parm, except that the indexes for each dimension can be specified. The indexes for unused dimensions should be specified as 0.

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	pointer to the internal parameter structure
buffer_length	length of the buffer, where the requested data has to be written to
buffer	address of buffer, where the requested data has to be written to. This buffer should be aligned to allow easy access to I2/I4/F4/F8 variables.
indexes	array with index information
return	Any value < 0 indicates an error during retrieval of the information. A value of -1 indicates an illegal parameter number. A value of -2 indicates an internal error. A value of -3 indicates that data has been truncated. A value of -4 indicates that data is not an array. A value of -7 indicates an interface version conflict. A value of -100 indicates that the index for dimension 0 is out of range. A value of -101 indicates that the index for dimension 1 is out of range. A value of -102 indicates that the index for dimension 2 is out of range. A value of 0 indicates successful operation. A value > 0 indicates successful operation, but the data was only this number of bytes long (buffer was longer than the data).

Write back operand data:

These functions are used by the 3GL program to write back the data to any parameter. Natural identifies the parameter by the given parameter number and writes the parameter data from the given buffer address with the given buffer size to the parameter data. If the parameter data is shorter than the given buffer size, the data will be truncated to the parameters data length, i.e., the rest of the buffer will be ignored. If the parameter data is longer than the given buffer size, the data will copied only to the given buffer length, the rest of the parameter stays untouched. This applies to arrays in the same way. For dynamic variables as parameters, the parameter is resized to the given buffer length.

If data gets truncated for variables of the type I2/I4/F4/F8 (buffer length not equal to the total parameter length), the results depend on the machine type (little endian/big endian). In some applications, the user exit must be programmed to use no static data to make recursion possible.

Prototypes:

```
int ncxr_put_parm        ( int parmnum, void *parmhandle,
                           int buffer_length, void *buffer );
  int ncxr_put_parm_array  ( int parmnum, void *parmhandle,
                           int buffer_length, void *buffer,
                           int *indexes );
```

Parameter description:

parmnum	Ordinal number of the parameter. This identifies the parameter of the passed parameter list. Range: 0 ... numparm-1.
parmhandle	pointer to the internal parameter structure.
buffer_length	length of the data to be copied back to the address of buffer, where the data comes from.
indexes	index information
return	Any value < 0 indicates an error during copying of the information: A value of -1 indicates an illegal parameter number. A value of -2 indicates an internal error. A value of -3 indicates that too much data has been given. The copy back was done with parameter length. A value of -4 indicates that the parameter is not an array. A value of -5 indicates that the parameter is protected (constant or AD=O). A value of -6 indicates that the dynamic variable could not be resized due to an 'out of memory' condition. A value of -7 indicates an interface version conflict. A value of -100 indicates that the index for dimension 0 is out of range A value of -101 indicates that the index for dimension 1 is out of range A value of -102 indicates that the index for dimension 2 is out of range A value of 0 indicates successful operation. A value > 0 indicates successful operation., but the parameter was this number of bytes long (length of parameter > given length)

All function prototypes are declared in the file natuser.h.

Part II: CALL under OpenVMS, UNIX and Windows

- Return Code
- User Exits under Windows
- User Exits under Open VMS
- User Exits under UNIX

Return Code

The condition code of any called function may be obtained by using the Natural system function RET.

Example:

```
...
RESET #RETURN(B4)
CALL 'PROG1'
IF RET ('PROG1') > #RETURN
  WRITE 'ERROR OCCURRED IN PROGRAM1'
END-IF
...
```

User Exits under Windows

Under Windows, user exits are needed to be able to access external functions that are invoked with a CALL statement. The user exits have to be placed in a DLL (dynamic link library). For further information on the user exits, please refer to the following file:

%NATDIR%\%NATVERS%\samples\sysexuex\readme.txt

User Exits under OpenVMS

Under OpenVMS, user exits are needed to be able to access external functions that are invoked with a CALL statement. The user exits have to be placed in a shareable image. For further information on the user exits, please refer to the following file:

NATSAMPLES:readme.txt

User Exits under UNIX

- Step 1 - Defining the Jump Table
- Step 2 - Writing the External Functions
- Step 3 - Compiling and Linking
- How to Build a Shared Library
- Using the Shared Library
- How to Generate a Static Nucleus
- Example Programs

Under UNIX, user exits are needed to make external functions available and to access operating-system interfaces that are not available to Natural.

The user exits can be placed either in a shared library and thus linked dynamically, or in a library that is linked statically to the Natural nucleus.

If they are placed in shared libraries, it is not necessary to relink Natural whenever a user exit is modified. This makes the development and testing of user exits a lot easier. This feature is available under all operating systems that support shared libraries.
Under all operating systems, it is possible to place user exits in a library that is linked to the Natural nucleus; that is, to statically link the user exits with the Natural prelinked object "natraw.o".

A user exit is added to Natural in three steps:

1. A jump table has to be created that allows Natural to associate the name of a function invoked by a CALL statement with the address of the function.
2. The functions that were put into the jump table must be written.
3. In the case of a dynamic link, the shared library that contains the user exits has to be rebuilt.
   In the case of a static link, the jump table and the external functions must be linked together with the prelinked Natural nucleus, to produce an executable Natural nucleus that supports the external functions.

Step 1 - Defining the Jump Table

A sample of a jump table - "jumptab.c" - can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

Step 2 - Writing the External Functions

Each function has three parameters and returns a long integer. A function prototype should be as follows:

```
NATFCT myadd  (nparm, parmptr, parmdec)
```

```
WORD  nparm;
BYTE  **parmptr;
FINFO  *parmdec;
```

nparm	16 bit unsigned short value, containing the total number of transferred operands (operand2).
parmptr	Array of pointers, pointing to the transferred operands.
parmdec	Array of field information for each transferred operand.

The data type FINFO is defined as follows:

```
typedef struct {
  unsigned char     TypeVar;      /* type of variable                        */
  unsigned char     pb2;          /* if type == ('D', 'N', 'P' or 'T') ==>    */
                                  /*    total num of digits                 */
                                  /* else                                    */
  union {                         /*    unused                             */
    unsigned char   pb[2];        /* if type == ('D', 'N', 'P' or 'T') ==>    */
    unsigned short  lfield;       /*    pb[0] = #dig before.dec.point      */
  } flen;                         /*    pb[1] = #dig  after.dec.point      */
                                  /* else                                  */
                                  /*    lfield = length of field           */
} FINFO;
```

Next, the module containing the external functions must be written. A sample function - "mycadd.c" - can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

Step 3 - Compiling and Linking

The file "natuser.h", which is included by the sample program, is delivered with Natural. It contains declarations for the data types BYTE, WORD and the FINFO structure, that is, the description of the internal representation of each passed parameter.

- In the case of dynamically linked user exits, the shared library containing the user exits has to be rebuilt.
- In the case of statically linked user exits, the Natural nucleus has to be relinked.

For these purposes, it is strongly recommended to use the sample makefiles supplied by Software AG, as they already contain the necessary compiler and linker parameters. The sample makefiles can be found in the directory:

$NATDIR/$NATVERS/samples/sysexuex

For further information, see the following sections and the explanations in the makefiles themselves.

How to Build a Shared Library

1. From the example directory, which is contained in
   $NATDIR/$NATVERS/samples/sysexuex
   copy the following files into your work directory:
   Makedyn
   jumptab.c
   ncuxinit.c
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file "jumptab.c" to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits. You might consider cutting and pasting the appropriate sections from your pre-2.2 version of "jumptab.c".
4. Edit the makefile as follows:
   Specify the names of the object files containing the user exits in the following line:
   USEROBJS =
   Specify the name of the resulting shared library in the following line:
   USERLIB =
   If you need to include private header files, specify the directories containing them in the following line:
   INCDIR =
5. To remove all unneeded files, issue the command:

make -f Makedyn clean

6. To compile and link your shared library, issue the command:
   make -f Makedyn lib

Using the Shared Library

Set the environment variable NATUSER to the libraries you want to use. For example:

setenv NATUSER $NATDIR/$NATVERS/bin/<library-name>

You must specify a full qualified path name for the shared library.

You can specify more than one path if you delimit them with a colon (:) like the UNIX PATH variable.

Example:

See the sample user exit function in **$NATDIR/$NATVERS/samples/sysexuex**.

Note:
The libraries are searched in the order in which they are specified in NATUSER. This means that if two libraries contain a function of the same name, Natural always calls the function in the library which is specified first in NATUSER.

How to Generate a Static Nucleus

1. From the example directory, which is contained in **$NATDIR/$NATVERS/samples/sysexuex**
   copy the following files into your work directory:
   Makefile
   jumptab.c
2. Copy the C source files which contain your user exits into the same work directory.
3. Edit the file "jumptab.c" to include the names and function pointers for your user exits. To do so, you add in Section 2 the external declarations of your user exits, and in Section 3 you add the name/function-pointer pairs for your user exits. You might consider cutting and pasting the appropriate sections from your pre-2.2 version of "jumptab.c".
4. Edit the makefile as follows:
   Specify the names of the object files containing the user exits in the following line:
   USEROBJS =
   If you need to include private header files, specify the directories containing them in the following line:
   INCDIR =
5. Issue the command "make" to get information about further processing options.

Example:

See the sample user exit function in **$NATDIR/$NATVERS/samples/sysexuex**.

Example Programs:

After successful compilation and linking, the external programs can be invoked from a Natural program. Corresponding Natural example programs are provided in the library SYSEXUEX.